Thread-Safe Reactive Programming

Drechsler, Joscha; Mogk, Ragnar; Salvaneschi, Guido et al.

(2018)

Thread-Safe Reactive Programming

JOSCHA DRECHSLER, Technische Universität Darmstadt, Germany
RAGNAR MOGK, Technische Universität Darmstadt, Germany
GUIDO SALVANESCHI, Technische Universität Darmstadt, Germany
MIRA MEZINI, Technische Universität Darmstadt, Germany

The execution of an application written in a reactive language involves transfer of data and control flow between imperative and reactive abstractions at well-defined points. In a multi-threaded environment, multiple such interactions may execute concurrently, potentially causing data races and event ordering ambiguities. Existing RP languages either disable multi-threading or handle it at the cost of reducing expressiveness or weakening consistency. This paper proposes a model for thread-safe reactive programming (RP) that ensures abort-free strict serializability under concurrency while sacrificing neither expressiveness nor consistency. We also propose an architecture for integrating a corresponding scheduler into the RP language runtime, such that thread-safety is provided "out-of-the-box" to the applications.

We show the feasibility of our proposal by providing and evaluating a ready-to-use implementation integrated into the REScala programming language. The scheduling algorithm is formally proven correct. A thorough empirical evaluation shows that reactive applications build on top of it scale with multiple threads, while the scheduler incurs acceptable performance overhead in a single-threaded configuration. The scalability enabled by our scheduler is roughly on-par with that of hand-crafted application-specific locking and better than the scalability enabled by a scheduler using an off-the-shelf software transactional memory library.

CCS Concepts: • **Theory of computation → Concurrent algorithms**; Dynamic graph algorithms; • **Computing methodologies → Parallel programming languages**; *Shared memory algorithms*;

Additional Key Words and Phrases: Reactive Programming, Concurrency, Transactions

ACM Reference Format:
Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Thread-Safe Reactive Programming. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 107 (November 2018), 30 pages. https://doi.org/10.1145/3276477

1 INTRODUCTION

Reactive Programming (RP) [Bainomugisha et al. 2013] features two kinds of *reactives*: Events and Signals. Events are composable streams that can emit values. Signals are composable self-updating computations with "spreadsheet semantics". The RP runtime tracks all dataflow dependencies between reactives to automatically propagate changes from inputs with *glitch freedom* consistency. This relieves developers from manually maintaining control flow, which in event-driven software is often obstructed by "callback hell" [Meyerovich et al. 2009]. As a result, RP enables modular and declarative applications with improved code quality and maintainability [Salvaneschi et al. 2014a].

Authors' addresses: Joscha Drechsler, Informatik, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, drechsler@cs.tu-darmstadt.de; Ragnar Mogk, Informatik, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, Mogk@cs.tu-darmstadt.de; Guido Salvaneschi, Informatik, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, salvaneschi@cs.tu-darmstadt.de; Mira Mezini, Informatik, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, mezini@cs.tu-darmstadt.de.

2475-1421/2018/11-ART107
https://doi.org/10.1145/3276477

This paper addresses thread-safety for RP. In a multi-threaded environment, multiple of the following three interactions between the imperative and reactive abstractions can occur concurrently. First, firing designated input Events, or reassigning Signal variables. Second, reading ("pulling") the value of Signals. Third, computations that define derived reactives executing side-effects to "push" changes into imperative code. Multiple of these interactions concurrently affecting common reactives leads to data races and ambiguities in that different concurrent changes arrive at different derived reactives in different orders. Thus, RP systems must have a clear stance in relation to multi-threading, preferably without sacrificing performance, consistency, or expressiveness.

No state-of-the-art systems succeed in providing all of these criteria. Scala.React [Maier and Odersky 2012] and Distributed REScala [Drechsler et al. 2014] ensure thread-safety by using a global lock to prohibit parallelism. "Containers and Aggregates, Mutators and Isolates" [Prokopec et al. 2014] do not address consistency even for single-threaded change propagation, and address thread-safety limited to the scope of individual Signals and Events. Scala.Rx [Haoyi 2016] and QUARP [Proença and Baquero 2017] only provide eventual consistency, in that derived Signals skip intermediate updates if concurrent changes interact. Moreover, both systems support only Signals, but not Events, since such a semantics of randomly skipping some values is not usable for Events. Elm [Czaplicki and Chong 2013] has support for parallelism in theory, but its use is limited in practice because it compiles to JavaScript, which has a difficult relation with parallelism. More importantly though, Elm does not support changing dependencies at run-time, thus restricting the language expressiveness. For example, clients joining and leaving an application can no longer be modelled using reactives. Our objective is a RP approach supports expressiveness and consistency in a multi-threaded environment, with Events and Signals, reactive computations with dynamically changing dependencies, and side-effects.

To achieve consistency, it is necessary to address combinations of multiple operations across several reactives as *transactions*. However, the common practice of aborting and redoing transactions is inapplicable due to side-effects in reactive computations. Schedulers for abort-free execution of transactions are notoriously difficult to conceive though, because they must guarantee success and correctness before allowing side-effects to be executed. The RP runtime possesses knowledge (e.g., tracked data dependencies) and control capabilities (e.g., control flow authority), and adheres to a well-defined, restricted execution model (i.e., glitch-free change propagation). We overcome these challenges by using unique opportunities of the RP model. We exploit these aspects in a scheduler that enables very fine-grained and efficient parallelization while ensuring abort-free and strict serializable execution of concurrent reactive updates.

In summary, our key contributions are as follows:

- We dissect the RP system into basic components, operations and processes, and analyze them for interactions that occur when they execute in a multi-threaded environment. Based on this analysis we define a model for thread-safe RP that ensures abort-free strict serializability for concurrent executions without sacrificing expressiveness or consistency (§3).
- We introduce a scheduling algorithm, called MV-RP, that enables concurrency-agnostic programming by implicitly executing every imperative interaction as one transaction, and give an intuition of how it achieves our proposed correctness (§4). A formalization and correctness proof for MV-RP are available in an extended technical report [Drechsler et al. 2018].
- We provide a ready-to-use implementation of MV-RP transparently integrated into the runtime of REScala [Salvaneschi et al. 2014b].[1] Our extensive performance evaluation (§5) shows that reactive applications executed with MV-RP scale with multiple threads, while the scheduler incurs acceptable performance overhead. The scalability enabled by MV-RP is roughly on par

[1]We chose REScala, because its API is designed to support different backend runtimes.

with that of hand-crafted application-specific locking, and better while also providing stronger guarantees than a scheduler using an off-the-shelf Software Transactional Memory library.

§2 introduces RP concepts from the user perspective, §6 positions our approach with respect to related work, and §7 concludes the paper and discusses areas of future work.

2 REACTIVE PROGRAMMING IN RESCALA

We introduce (single-threaded) RP from the developer's perspective by implementing an example application in REScala. Our design is inspired by the dining philosophers [Hoare 1985], with philosophers sharing forks around a table. We later introduce concurrency and discuss its effects through multiple threads concurrently executing updates for different philosophers. For flexibility in later examples and benchmarks, the table's `SIZE` (number of philosophers and forks) is parametric.

2.1 The User Perspective

We introduce Events, Signals, conversions between them and their imperative interactions.

Signals. Philosophers are modelled as Vars of type `Phil`, initially in state `Thinking`.

```
1 cases Phil = { Thinking, Eating }
2 val phils = for (j <- 1 until SIZE) yield Var[Phil](Thinking)
```

A REScala `Var` is a kind of mutable variable. Like ordinary variables, the value of a `Var` can be read (through `now`, e.g., `phils(1).now` reads the current value of the philosopher with index 1) and overwritten (through `set(...)`, e.g., `phils(1).set(Eating)`). Unlike ordinary variables, changes of `Vars` are automatically propagated to self-updating *derived* Signals that use the `Vars` in their definition. In the following, forks are also modelled as enumerations with two states. But, unlike philosophers, they are implemented as derived Signals.

The `Signal` keyword instantiates a derived Signal (here of type `Fork`) given a user-defined computation, akin to a spreadsheet formula. The computation can *depend on* (the `.depend` keyword[2]) values of multiple other Vars and/or Signals, its *dependencies*. In our example (Line 7), each fork's value is derived from the values of the philosopher with the same index and the next circular index. If both are `Thinking` (Line 8), the fork is `Free`. If one philosopher is `Eating` (Lines 9 and 10), the fork is taken. Otherwise (Line 11), the fork raises an error (can't be used by two philosophers simultaneously).

```
3 cases Fork = { Free, Taken(by: Int) }
4 val forks = for (idx <- 0 until SIZE)
5   yield Signal[Fork] {
6   val nIdx = (idx + 1) % SIZE
7   (phils(idx).depend, phils(nIdx).depend) match {
8     case (Thinking, Thinking) => Free
9     case (Eating, Thinking) => Taken(idx)
10    case (Thinking, Eating) => Taken(nIdx)
11    case (Eating, Eating) =>
12      throw new AssertionError()  } }
```

Like Vars, derived Signals' current values can be read imperatively (e.g., `forks(0).now`), but unlike Vars, not set. Instead, after the value of any dependency changed, their values are updated automatically through *reevaluation*, i.e., re-execution of their computation. E.g., upon `phils(1).set(Eating)`, both `forks(0)` and `forks(1)` will be reevaluated and change their values to `Taken(1)`.

Each fork has static dependencies on the same two philosophers at all times. In general though, Signals may have *dynamic* dependencies. Below, `Sight` (line 13) represents philosophers' possible perception of their forks – respective `sights` Signals are instantiated in line 14. Each `sight(i)` first depends on the philosopher's left fork (Line 17) to distinguish three cases.

[2]In the REScala API, `n.depend` is abbreviated as `n()`; we use the explicit `.depend` notation here for naming consistency.

```
13 cases Sight = { Ready, Done, Blocked(by: Int) }
14 val sights = for (idx <- 0 until SIZE)
15   yield Signal[Sight]{
16   val prevIdx = (idx - 1 + SIZE) % SIZE
17   forks(prevIdx).depend match {
18     case Free => forks(idx).depend match {
19         case Taken(neighbor) => Blocked(neighbor)
20         case Free => Ready }
21     case Taken(`idx`) =>
22       assert(forks(idx).depend == Taken(idx))
23       Done
24     case Taken(neighbor) => Blocked(neighbor) } }
```

(1) Left fork is `Free`. Then, `sight` also depends on the right fork (Line 18). If the latter is `Taken` (Line 19), `sight` is `Blocked` with neighbor's index; otherwise (Line 20), `sight` is `Ready`. (2) Left fork is `Taken` by the philosopher himself (Lines 21 to 23). Then `sight` is `Done` (he himself is eating). In this case, `sight` also depends on the right fork (Line 22) to assert that it is also taken by the philosopher himself. (3) Left fork is `Taken` by a neighbor (Line 24). Then `sight` shows `Blocked` by that neighbor and does *not* depend on the right fork. The dependency of `sight` on the right fork is dynamic; it is *discovered* or *dropped* when a reevaluation of `sight` switches out of or into the last case. After the dependency was dropped, changes of the right fork's value do not trigger reevaluations of `sight` until the dependency is re-discovered again.

Dynamic dependencies enable important features. One example is the creation and removal of new Signals at run-time, which must be newly (dis-)connected with their dependencies. Another example are *higher-order* Signals, i.e., Signals whose value contains pointers to other (*inner*) Signals.

Events and Conversions. Unlike Signals, Events only occasionally emit a value (e.g., mouse clicks) and thus cannot be read imperatively. Analogously to input Vars and derived Signals, there are input Events (not shown), which can be fired imperatively through `evt.fire(value)`, and derived Events, which depend on other reactives and may emit values when reevaluated. Events and Signals can also be converted to and derived from each other, as shown below.

```
25 val sightChngs: Seq[Event[Sight]] = for (i <- 0 until SIZE) yield sights(i).changed
26 val successes = for (i <- 0 until SIZE) yield sightChngs(i).filter(_ == Done)
27 val counts: Seq[Signal[Int]] = for (i <- 0 until SIZE) yield
28                                  successes(i).fold(0) { (acc, _) => acc + 1 }
```

The `.changed` derivation converts a Signal (`sights(i)`) into an Event that emits each new value of the Signal. The `.filter` derivation forwards emitted values only if they match a predicate; here, `successes` Events fire whenever the respective `sightChngs` emit `Done`. The `.fold` derivation converts the `successes` back into Signals. Folding Signals accumulate all values emitted by an Event, similar to folding over (infinite) lists. Here, each folding `counts` Signals start with an initial value of `0`, which is incremented whenever the respective `successes` Event fires, i.e., they count the philosopher's successful `Eating` iterations.

Interactions with the Environment. So far we have discussed input reactives that can be imperatively set (`evt.fire(value)`, resp. `var.set(value)`) to start reactive updates, and that Signals' values can be imperatively read (`s.now`) to "pull" updated values out of RP abstractions. As a third and final interaction between imperative and reactive abstractions, Event and Signal computations can contain *side-effects* to "push" changes back into imperative reactions.

```
29 val totalCount = successCounts.reduce(_ + _)
30 val output = totalCount.map{v => println(... v ...)}
```

We `reduce` (a method of Scala's collection library, not related to RP) all `successCounts`, adding them together one by one into the `totalCount` Signal, which thus counts overall successful `Eating` iterations. We then `map` the values of `totalCount` using the `println` function. `s.map(f)` is shorthand for `Signal{f(s.depend)}`, and usually derives a new Signal by transforming values of `s` via function

`f` (exists equally for Events). In this case though, `println` is a void method that always only returns the `Unit` value, i.e., the resulting Signal's value is useless. But, during reevaluations, it executes the side-effect of printing the value (`v`) of `totalCount` to console.

2.2 RP Behind the Scenes

Dependency Graph. The runtime representation of a RP program is its directed acyclic *dependency graph (DG)*, built while the execution of a reactive program unrolls. Fig. 2 (a) shows parts of the philosophers' DG. Nodes represent reactives and arrows their dependency relations pointing in the direction of the data flow. A `phils` Var is visualized as a triangle labelled with its index at the bottom, two `forks` and one `sights` derived Signal are visualized as circles.

Change Propagation. While the RP system is quiescent and all Signals' values are up-to-date in terms of the input values of their computations, we say the DG is in a *consistent resting state*. Imperative changes (`set(...)` and `fire(...)` calls) cause Events and Signals react and propagate these changes derived reactives. E.g., consider a thread that continuously tries to switch a philosopher between `Thinking` and `Eating`. Fig. 1 exemplifies such a loop for `phils(1)`. We visualize reevaluated nodes through shading and changed values through bold font. E.g., in Fig. 2 (b) the `phils(1)` Var has been changed to `Eating`. Change propagation moves outwards over connected edges from every changed node, and every reached derived reactive will be reevaluated. We visualize change propagation through bold edges. E.g., in Fig. 2 (c), above change has been propagated to `forks(0)` which thus reevaluated to `Taken(1)`. After all changes have been propagated and all reevaluations completed, we say the DG has reached an updated consistent resting state. Each imperative input change therefore causes a change propagation, which takes the DG from one consistent resting state to the next.

```
1 val idx = 1
2 while(System.in.available == 0) {
3   if (sights(idx).now == Ready)
4     phils(idx).set(Eating)
5   if (sights(idx).now == Done)
6     phils(idx).set(Thinking)
7 }
```

Fig. 1. Driving thread for philosopher 1

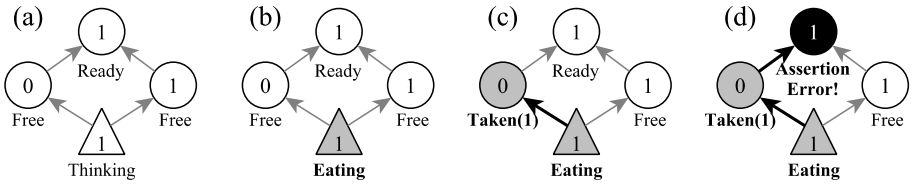


Fig. 2. Section of a philosophers dependency graph with change propagation and a glitch.

Glitch Freedom. RP systems provide a consistency guarantee called *glitch freedom*, i.e., the absence of *glitches*. Glitch freedom means, that every reevaluation is executed such that all values it reads correspond to the updated consistent resting state of the DG. Under glitch freedom, computations can not observe a set of values that some of which have been updated by the in-progress change propagation, but some will still be updated later. This ensures, that reevaluations can rely upon the invariants that exist between reactives in the consistent resting states of the DG. Fig. 2 (d) visualizes a glitch that breaks the invariant that a philosopher always holds either both or none of his forks. The change of `forks(0)` has propagated to and triggered a reevaluation of `sights(1)`. Its computation then executes the case of `forks(0)` being `Taken(1)` by the philosopher himself (Line 21 in §2.1) and the associated assertion (Line 22). This assertion fails: `forks(1)` is still `Free`, as it has not yet been reevaluated after the change of `phils(1)`. This example is synthetic, but equally grave errors may occur in realistic applications with similar invariants, e.g., when combining a list Signal with an index Signal whose value glitches outside the list's bounds.

To achieve glitch freedom, RP systems use *propagation algorithms* that order reevaluations so that glitches do not occur. Most propagation algorithms compute the reevaluation order as a topological order of the DG. The order in Fig. 2 is not glitch-free, because it is inconsistent with the DG topology. In general, these orders need to be devised on-the-fly, since any pre-computed order may become incorrect when the DG topology changes due to dynamic dependencies.

3 REACTIVE PROGRAM EXECUTIONS UNDER CONCURRENCY

We introduce concurrency into our philosophers running example by, in essence, spawning one driver thread from Fig. 1 for each philosopher on one table. In general, when a RP system is embedded into a multi-threaded environment, an arbitrary number of threads can concurrently read Signals, change Vars, and/or fire input Events. Concurrent input changes trigger concurrent reevaluations of derived reactives, which in turn result in concurrent reads of their parameters' values and updates to the derived reactives' values. It is easy to see that concurrent read and write operations on the same reactive may result in race conditions, where the read may execute either before or after the write. But noticeably more complex issues arise as well. Multiple concurrently executing change propagations may have more than one reactive in common, that they will reevaluate. If they reach and reevaluate one of these in a different order than the other, it can become impossible for yet other reactives to be reevaluated glitch-free. In this section, we contribute a systematic account of the issues introduced to RP when adding concurrency.

To systematically study the subject matter, we first lay out the anatomy of a RP system – its components and the operations through which they interact to implement reactive change propagation. Next, we use the anatomy to exemplify increasingly complex race and ordering conditions between concurrent operations, and derive restrictions on their interleaving that are necessary to achieve thread-safety. Finally, we conclude with a formal correctness definition for the execution of the operations.

3.1 The Anatomy of a Reactive Programming System

The anatomy of a RP system is depicted in Fig. 3. It consists of the following components. The "imperative environment" in the lower-left corner represents user-defined threads and regular variables of the host programming language (e.g., the loop in Fig. 1). The "event/signal computations" in the top-left corner represent the user-defined computations of derived reactives (e.g., all code from §2.1). The DG of all reactives and their dependency relations is depicted in the top-right corner. Finally, the propagation algorithm is shown in the lower-right corner.

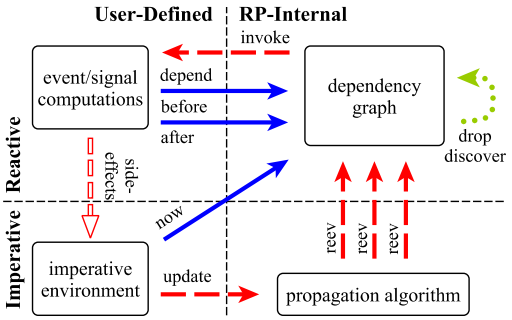


Fig. 3. Operations between RP Components

The arrows in Fig. 3 visualize the operations through which the components interact. We distinguish three categories of operations, which we visualize through different kinds of arrows: (a) reading the values of DG nodes (solid blue arrows), (b) propagating changes through the DG (dashed red arrows), and (c) performing dynamic changes of DG edges (dotted green arrows). As we elaborate later in this section, these are the operations whose interleaved executions we need to restrict to ensure thread-safety of RP executions. Hence, in the following, we map all RP features to these operations and explain each of them in detail, including the data they access and how this data is stored in the DG's nodes.

Operations for reading DG node values (solid blue arrows in Fig. 3) are `now`, `after`, `before`, `depend`. The simplest one is `s.now` (`s` is a Signal), which reads and returns the current value of `s`. When a derived reactive `d` reevaluates, its user computation, $comp_d$, may also read the value of other reactives, e.g., `n`, which thereby become its parameters. Reading parameters happens through `n.before` or `n.after`, instead of `n.now`, because `n.now` is ambiguous in this context. If `n` is also affected by the change propagation that reevaluates `d` and no (transitive) dependency between `d` and `n` exists, `n` may be reevaluated before or after `d`, i.e., `n.now` may return the value of `n` before or after it is reevaluated. In most cases, $comp_d$ reads the glitch-free value of its parameters through `n.after` – these calls may possibly suspend until `n` is glitch-free.

The simplest use of `before` is in a folding Signal `f`: To compute the updated value of `f`, $comp_f$ must first read the old value of `f` through `f.before`. Folding Signals thus implement self-loops on individual DG nodes. Calling `before` on other Signals further down the DG topology generalizes this pattern to "backwards edges" that close cycles across multiple DG nodes (cf. `last` keyword in [Sawada and Watanabe 2016]). But, there are also use cases for `before` not related to DG cycles. E.g., a `submit` Event in a chat GUI reevaluates the `text` input field to remove the sent text, but simultaneously needs to emit a `message` Event to the server with the value of `text` from *before* it was cleared. On Events `e`, only the `e.after` operation is available for reading their value. This is because Events only have an emitted value after they were reevaluated, but not before (`before`) and not outside of change propagation (`now`).

Most parameter reads inside any $comp_d$ are `n.depend`. E.g., all reads in all $comp_d$ in §2.1 are `n.depend`, except for the folding `counts(i).before` self-call. This includes all internally executed reads by any `changed`, `filter`, `map`, etc. derivations. The `n.depend` operation behaves identical to `n.after`, but additionally registers `d` for reevaluation upon changes of `n`. The choice between `n.depend` and `n.after` thus grants all $comp_d$ fine-grained control over which `d` are dependencies or just parameters, i.e., which values' changes do or do not trigger the next reevaluation of `n`.

Operations for implementing change propagation through the DG (solid red arrows in Fig. 3) are `update`, `reev` (short for reevaluate), and `invoke`. Change propagation is always initiated by an imperative `update(i1 -> v1, i2 -> v2, ...)` call. Calls `var.set(v)` or `evt.fire(v)` are shorthand for `update(var -> v)` or `update(evt -> v)`. `update(...)` calls do not directly update the values of the given input nodes, but are dispatched through the propagation algorithm. The latter translates them into a series of `reev` calls on DG nodes to propagate the input changes glitch-free through the entire application. The execution of each `reev` involves the respective DG nodes' values and variables, which are depicted as grey boxes (values) and white boxes (variables) in Fig. 4, and behaves differently depending on the reactive.

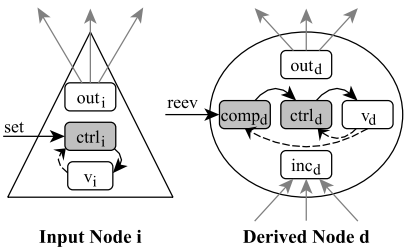


Fig. 4. Composition of reactive nodes.

Input nodes (depicted on the left of Fig. 4) consist of variables v_i and out_i and *control code* $ctrl_i$. v_i holds the emitted/stored value of i and out_i its *outgoing dependencies*, i.e., the set of derived nodes currently registered on i for reevaluations. Initially, for each input `i` in `update(..., i -> v, ...)`, the propagation algorithm passes the corresponding `v` to $ctrl_i$. For Events, $ctrl_i$ always stores `v` as the emitted value v_i and always reports `i` as changed. For Signals, $ctrl_i$ writes `v` to v_i only if it is different from the previous v_i (this `before` self-call is visualized in Fig. 4 by the thin dashed arrow from v_i to $ctrl_i$). Together with the whether or not `i` changed, $ctrl_i$ returns out_i to the propagation algorithm. If `i` changed, the propagation algorithm must eventually reevaluate

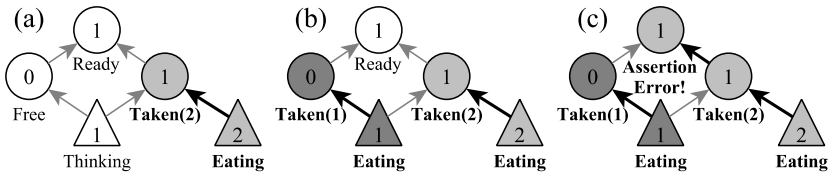


Fig. 5. A glitch from concurrent propagations interacting.

each reactive in out_i, once it can guarantee glitch-freedom. If `i` did not change, reevaluations of nodes in out_i triggered by other changes no longer need to wait for `i`.

Derived nodes `d` (depicted on the right of Fig. 4) consist of the same building blocks as input nodes, plus the user computation $comp_d$ and variable inc_d that stores *incoming dependencies* – the set of nodes on which `d` is currently registered for reevaluation.[3] Their control code $ctrl_d$ for Events and Signals is identical to input nodes (determine if changed, write v_d and additionally return out_d). However, the value `v` passed to $ctrl_d$ is not taken from the `update(...)` call's parameters, but computed as the return value of (re-)executing $comp_d$. While $comp_d$ executes, the set of nodes n on which it calls `n.depend` is recorded. After $comp_d$ returns, but before the returned `v` is passed to $ctrl_d$, `d` updates its predecessor edges in DG to match that set. The edges to add or remove are computed as the differences between inc_d and the newly recorded set of `n.depend` nodes. If $comp_d$ executed `n.depend` but $n \notin inc_d$, `d` executes `n.discover(d)`. If $n \in inc_d$ but $comp_d$ did not execute `n.depend`, `d` executes `n.drop(d)`. Afterwards, the new set of `n.depend` nodes is written to inc_d. Only then, the execution of `reev` is finally completed by passing the `v` returned by $comp_d$ to $ctrl_d$.

Operations for dynamic changes of DG edges (dotted green arrows in Fig. 3) are `n.discover(d)` and `n.drop(d)`. They are executed between pairs of nodes inside the DG, and thus visualized as a loop on the DG component. The operation `n.discover(d)` updates $out_n := out_n \cup \{d\}$ (edge $n \longrightarrow d$ is discovered), while `n.drop(d)` updates $out_n := out_n \setminus \{d\}$ (edge $n \longrightarrow d$ is dropped).

3.2 Race Conditions between RP Operations

We first discuss race conditions between operations on the same reactive (i.e., same DG node). Consider two threads, each executing an `Eating` update on a philosopher, from the applications initial state (i.e., all `counts` are 0). Assume they race such that each increases the value of one of the two dependencies of the `totalCount` Signal to 1, and then both simultaneously reevaluate `totalCount`. Both reevaluations read both updated inputs, and thus both compute the latest value for `totalCount` as 2. Both simultaneously read the previous value of `totalCount` as 0, and thus both update it to 2 and both report `totalCount` to have changed. In response, both reevaluate `output`, and thus the application outputs 2 as the new total `Eating` count twice – a result that would not be possible under single-threaded execution.

This problem occurred because of a race condition between both threads doing a compare-then-update on $v_{totalCount}$. To prevent this, such compare-then-update operations on node's values must execute mutually exclusively. Note that, e.g., folding Signals generalize this problem, in that they read the node's own value (`before` self-call) already at the very beginning of their reevaluations. Thus, reevaluations as a whole must execute mutually exclusive on each node. With mutually exclusive reevaluations, the second thread's reevaluation of `totalCount` would always read the previous value as 2, thus determine the node as unchanged, and thus the new value 2 would correctly be printed only once instead of twice.

Mutually exclusive reevaluations would only resolve half of the problem at hand, though. When executing single-threaded, the application would always first output a new `totalCount` of 1, and

[3]The sets inc_d and out_n mirror each other in that every edge $n \longrightarrow d$ corresponds to $d \in out_n$ and $n \in inc_d$.

not jump straight to 2. But, in the concurrent example this is skipped, because the first thread's reevaluation already encompassed both threads' changes instead of just its own. Some may consider this issue harmless in this particular instance, but taking into account its more general implications shows that it is not. If `totalCount` were an Event instead of a Signal, this same issue would result in the Event spuriously skipping some values, which is not tolerable. Moreover, in more complex situations even with just Signals, other thread's changes may be observed only partially, which then causes a glitch even if all threads by themselves follow a glitch-free reevaluation order. Fig. 5 demonstrates this, showing that also race conditions between operations across on multiple reactives must be considered to ensure safe semantics.

The graph section shown in Fig. 5 is the same as in Fig. 2, but with `phils(2)` included additionally. In (a), thread T_2 executes `phils(2).set(Eating)` and reevaluates `forks(1)` to `Taken(2)` (nodes shaded light grey). In (b), thread T_1 reads `sights(1)` as `Ready`, changes `phils(1)` to `Eating`, and reevaluates `forks(0)` to `Taken(1)` (nodes shaded dark grey). T_1 has not yet reevaluated `forks(1)`, hence `sights(1)` is not ready for a glitch-free reevaluation. In (c), T_2 propagates its change of `forks(1)` to `sights(1)`. From the local perspective of T_2, both predecessors of `sights(1)` are now glitch-free: `forks(0)` is not affected by the propagation of T_2, and the reevaluation of `forks(1)` is already completed. Thus, unaware of the (partial) changes by T_1, T_2 deems `sights(1)` ready for a glitch-free reevaluation. But this is incorrect, because the changes by T_1 are incomplete, and the reevaluation produces the same glitch we demonstrated in Fig. 2 (`sights(1)` fails its assertion). Besides reevaluations executing mutually exclusive on each reactive, each reevaluation thus must also be isolated against reevaluations of all other reactives which it reads or depends upon.

Unlike for reevaluations on individual nodes, ad-hoc mutual exclusion is insufficient to synchronize reevaluations across multiple nodes, because deadlocks may occur. Fig. 6 demonstrates this on a table of only (for simplicity) two philosophers and forks. The two associated threads simultaneously `set(Eating)` (nodes are shaded light, respectively dark, grey), and propagate this to their respective left fork. Each fork should reevaluate, but each already has one dependency reevaluated by one thread, but the other by the other thread. Thus it is already too late to establish any kind of mutual exclusion – the application is deadlocked.

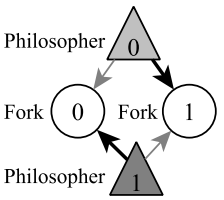


Fig. 6. A deadlock.

It is common practice in, e.g., databases, to resolve such deadlocks by aborting one of the involved parties, undo its changes, and restart it. Since we want to support side-effects in reactive user computations though, which we cannot undo, this solution is inapplicable here. Instead, synchronization between reevaluations of different nodes by different threads must be established preemptively, before any are executed, to ensure that deadlocks do not occur in the first place.

3.3 Correctness Property for Thread-safe Concurrent RP

§3.2 showed that concurrently executing the operations between RP components can result in harmful race conditions, and that preventing them requires complex synchronization to restrict their interleaving. To devise and verify corresponding scheduling algorithms, we need a correctness property for the execution of concurrent change propagations, which we derive next.

The most intuitive of these properties is *linearizability* [Herlihy and Wing 1990], where each operation takes effect atomically at some *linearization point* during its execution. From the perspective of the imperative environment, linearizability is desirable: all `n.now` and `update(...)` calls should behave and appear as if they executed atomically.

But, linearizability addresses only individual operations on singular data structures. It is therefore not applicable from the perspective inside the RP runtime or when taking into account user computations on reactives. Each node in the DG is a separate data structure and external `update(...)` calls

involve several $comp_d$, which execute several `s.before`, `n.after` and `n.depend` calls. To achieve a single linearization point for an `update(...)` call, all its reads and writes across all nodes must be orchestrated in some way to appear to take effect atomically together.

Transaction processing [Bernstein et al. 1986] provides a suitable model. Any number of reads and writes, which should appear as one atomic unit of work, are grouped into one transaction. Applied in the RP setting, every `s.now` call thus is a transaction containing a single read, and every `update(...)` call is a transaction that contains all the reads and writes of all reevaluations executed by the corresponding change propagation. Transaction processing provides a well-established formal tool set, to describe and verify the correctness of transaction systems. During execution, a transaction system produces a *history* – a sequence of all operations of every transaction in the order of their execution. Different levels of correctness of transaction systems are defined in terms of permutations of these histories that a system can produce.

Serial histories are those where all transactions execute without interleaving of operations. Systems that produce only serial histories are trivially correct, but very restrictive since they do not allow any parallelism. A less restrictive property is built on *view-equivalence* defined over *reads-from relations* of the form "transaction `A` reads the value of variable `x` that was written by transaction `B`". Two histories from the same set of transactions are *view-equivalent* if they are permutations of each other, but still have identical reads-from relations. Hence, view-equivalence implies that all reads of all transactions return the same value, i.e., transactions cannot distinguish between view-equivalent histories. Therefore, any histories that are view-equivalent to serial histories are equally acceptable as correct; such histories are called *serializable* histories.

Yet, serializability alone is not enough. A `s.now` transaction T_{now} could read a value of Signal `s` that was written by T_{past} half an hour earlier, and was already overwritten shortly after by another $T_{past'}$. This would conform with serializability, as T_{now} can formally be ordered between the T_{past} and $T_{past'}$. Yet, this would confuse users, as one would naturally expect such a read to return a current, up-to-date value of `s`. The foundation for a correctness definition for RP is thus the combination of serializability *and* linearizability, known as *strict serializability*: Transactions must execute their operations in a serializable fashion, and at the same time must have a linearization point within the actual real-time window of their execution.

To increase flexibility without loss of correctness guarantees, we make the following adaptations to strict serializability, based on various properties of RP.

(1) Usually, schedulers consider transactions of reads and writes on individual variables. The discussion in §3.1 revealed, that all variables ever accessed by the RP system are stored inside DG nodes. Moreover, all operations presented there (`now`, `reev`, `before`, `after`, `depend`, `drop` and `discover`) affect variables of single nodes only, and hence are easily executed linearizable per node, e.g., through simple mutual exclusion using Java's `synchronized` scope on the node instance. Thus, we design schedulers in terms of these more complex operations. This allows us to address scheduling solutions at a higher level of abstraction.
(2) We require strict serializability only for user-visible operations, i.e., those whose arrows in Fig. 3 have one end on the left-hand side. Other operations (e.g., `drop` or `discover`) may be executed arbitrarily, as long as they add up such that all user-visible operations are executed correctly. This gives schedulers more freedom to act in order to provide their guarantees, without changing users' experience.
(3) While a given propagation algorithm may be able to produce only a subset of all correct reevaluation orders, other reevaluation orders still remain correct. We thus accept any glitch-free order of reevaluations as correct, even if an underlying propagation algorithm is unable produce this order in a single-threaded setting. This makes it possible to devise and verify

the correctness of scheduling algorithms independently of concrete propagation algorithms. Further, it also gives more freedom to schedulers by allowing them to disregard and overrule propagation algorithms' reevaluation orders, as long as they still preserve glitch freedom.

(4) Lastly, recall that transactions of RP updates may contain side-effects, and thus cannot be aborted to resolve deadlocks. Because such aborts have become a common, widely accepted, and often even expected practice in many domains (e.g., databases, STM), we consider it necessary to explicitly state *abort-free* as an additional criterion for correctness.

Correctness property: Summarizing, we therefore define *correctness for the execution of concurrent RP transaction* as abort-free strict serializability for user-visible operations (`now`, `update`, `reev`, `after`, `before`, `depend`) with reevaluations of every update being executing in any glitch-free order.

4 MV-RP: A SCHEDULER FOR THREAD-SAFE REACTIVE PROGRAMMING

We extend the RP system in Fig. 3 with a scheduler that controls the execution of the operations from 3.1 to achieve the correctness property stated in 3.3. In this section, we describe the scheduler and intuitively indicate, how it produces abort-free strict serializable histories. We also show how we integrate the scheduler into the RP anatomy such that it is transparent to the application.

4.1 RP Architecture with an Integrated Scheduler

As already noted, to ensure thread-safe executions of reactive programs in multi-threaded environments, it suffices to protect accesses to the DG. The scheduler can achieve this result by intercepting all operations that act on the DG, including `n.discover(d)` and `n.drop(d)` which nodes inside the DG invoke on each other. Additionally, the scheduler also intercepts all `update(...)` calls by the imperative environment before they reach the propagation algorithm. While `update(...)` calls do not directly access any data stored in the DG, intercepting them allows the scheduler to (a) automatically wrap all operations triggered by each `update(...)` into one transaction, and (b) to proactively prepare for these operations, rather than merely being able to react once they happen, which is essential to support abort-free execution.

Fig. 7 shows a rearranged view of the components in Fig. 3 with the DG at the bottom and the scheduler added between it and the other components. All operations involving the DG are diverted through the scheduler. As a result, the scheduler can overrule decisions of the propagation algorithm regarding reevaluations (`reev`) when necessary to achieve correctness, as long as it retains glitch freedom.

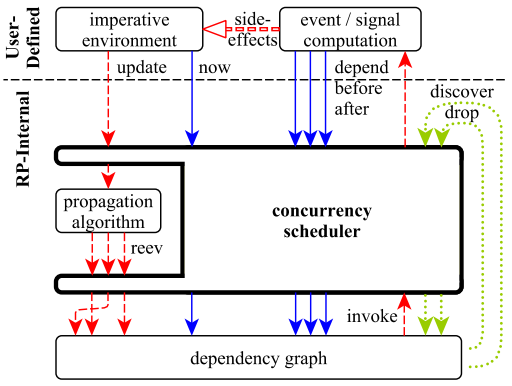


Fig. 7. Layered view of Fig. 3 with added scheduler.

The scheduler necessarily introduces additional operations on DG nodes (e.g., acquiring/releasing locks), which are invisible to the user code, hence irrelevant for correctness. In addition, by implementing the scheduler and propagation algorithm as a single integrated system, it is equally feasible to also introduce new operations between these two components. This allows them to cooperate instead of having the scheduler independently overruling the propagation algorithm's decisions. Specifically, our scheduler comes with a dedicated interface that propagation algorithms must implement to queue additional reactives for reevaluation with subsequent propagation, or to drop already queued ones.

4.2 The Scheduling Algorithm

Main building blocks. Our scheduler intertwines three synchronization techniques:

- **C2PL.** *Conservative two-phase locking* (C2PL) [Bernstein et al. 1986] provides abort-free strict serializability, but only if transactions can declare required resources prior to execution. Change propagation transactions (triggered by `update(...)` calls) can traverse the DG from all inputs that are about to change to reach all nodes that will be potentially reevaluated. Hence, C2PL can be used during `update(...)` calls to protect the execution of reevaluations (dashed red arrows in Fig. 3/7). However, C2PL alone is not enough to ensure abort-free strict serializability of update transactions due to reads and dynamic edge changes, as we elaborate next.
- **MVCC.** Unlike reevaluations, read operations (solid blue arrows in Fig. 3/7) target arbitrary nodes. The scheduler cannot predict these nodes ahead of time and thus cannot guard reads of their values through C2PL. Instead, we employ *multiversion concurrency control* (MVCC) [Bernstein et al. 1986], which gives the scheduler its name, MV-RP (Multi-Version-RP). MVCC maintains a backlog of past values for each variable (called *versions*), which in our combination are each written under control of C2PL. Because reads are idempotent, MVCC can execute reads "in the past" by returning old values from the backlog if necessary (while reads "in the future" are blocked until C2PL releases the corresponding version). This way, MVCC provides abort-free serializable execution, and with some additional care also strict, for pure read operations. Since writes are not idempotent, they cannot be executed in the past, meaning the combination of C2PL for reevaluations and MVCC for reads is indispensable for providing abort-free strict serializable execution for both kinds of operations.
- **Retrofitting.** Finally, dynamic dependency edge changes (dotted green arrows in Fig. 3/7) remain to be guarded. Dynamic dependency changes write the addition/removal of the discovered/-dropped DG edge on the edge's source node. They thus have an effect and are not idempotent, and therefore can not be executed "in the past" through MVCC. Like reads though, the scheduler cannot predict the source nodes of edge changes ahead of time, and therefore C2PL is inapplicable, too. Instead, we control their execution through *retrofitting*, a custom technique that provides abort-free strict serializable execution of dynamic edge changes by enabling their execution "in the past" despite their writes having an effect. Retrofitting is tightly intertwined with change propagation, MVCC, and C2PL. It becomes active when a mismatch occurs between when a dynamic edge change is executed in real time versus formal (serializability) time. Retrofitting exploits a synergy between RP's glitch freedom property and properties of the combination of MVCC and C2PL above. It executes ill-timed writes by "rewriting the history" of other nodes of the DG, re- or un-doing effects as necessary to achieve correct execution and maintain all of the guarantees provided by MVCC and C2PL.

The three techniques operate on a common data structure, called *node version history*. Each DG node has its own history, which associates a *node version* with each transaction that affected the node in some way. Each node version stores values for several variables of the node (cf. white components in Fig. 4). Version histories are sorted by a *serialization order* of transactions. The scheduler incrementally tracks this order as the directed acyclic *stored serialization graph* at runtime (*serialization graph testing* [Bernstein et al. 1986]). It is a partial order globally, but a total order over all versions within each node's history. Creating a version on a node requires the transaction to ensure that it is ordered against the transactions of all other versions that already exist on this node. To avoid creating cycles in the serialization order due to race conditions, the stored serialization graph provides an atomic (i.e., linearizable) "insert edge between two transaction nodes unless a path already exists" operation.

We elaborate on the scheduling algorithm, showing example updates on a section of the philosopher DG in Fig. 8. The figure shows several consecutive scheduling steps (a–e). The section of the dependency graph is shown in the middle part of each step. It consists of philosopher 1's input Var `phils(1)` (p_1), his neighbor Vars (p_0 and p_2), both his `forks` (f_0 and f_1), his `sights(1)` (s_1) and `totalCount` (c). Since s_1 is currently `Blocked(0)`, it does not have a dependency on f_1. Still, both s_1 and f_1 have paths to c, which are indicated across a squiggly lines cut-out that represents the omitted intermediate nodes on these paths. The version history of each node is visualized as a rectangle attached to the node via a dashed line. Each line of text in these rectangles displays one node version. The lines starting with label `0` represent the initial state for this example – both p_0 and p_2 are `Eating`, p_1 is `Thinking`. Other lines are labelled by the index of the transaction that placed the respective version. We denote transaction x as t_x, including the initial state as t_0. We do not visualize the transaction order of the stored serialization graph. Instead we crafted the example such that once transactions are ordered, their order matches their indices (i.e., t_1 before t_2 etc.).

Framing Phase. Before executing, all transactions perform C2PL during the *framing phase*. Fig. 8 (a) demonstrates C2PL in the multiversion environment. This step represents a state of the application, where two threads concurrently admitted changes of p_0 and p_2 to `Thinking`. The corresponding propagations are executed as transactions t_1 and t_2. In the framing phase, every transaction (e.g., t_1) traverses the DG from all changed inputs (e.g., p_0) to acquire *placeholder versions* on every node.

Placeholders represent planned reevaluations and a queue for the respective node's exclusive lock. Only the transaction with the first placeholder is allowed to reevaluate a given node. Fig. 8 visualizes placeholder versions as long underlines without any text; newly created versions in each step are highlighted with a grey background. In step (a), t_2 has created placeholders on p_2 and f_1, but not yet on c, and thus is still in the process of creating more. t_1 has created placeholders on all nodes from p_0 through c and thus has fulfilled the first of two conditions to switch from framing to executing. The second

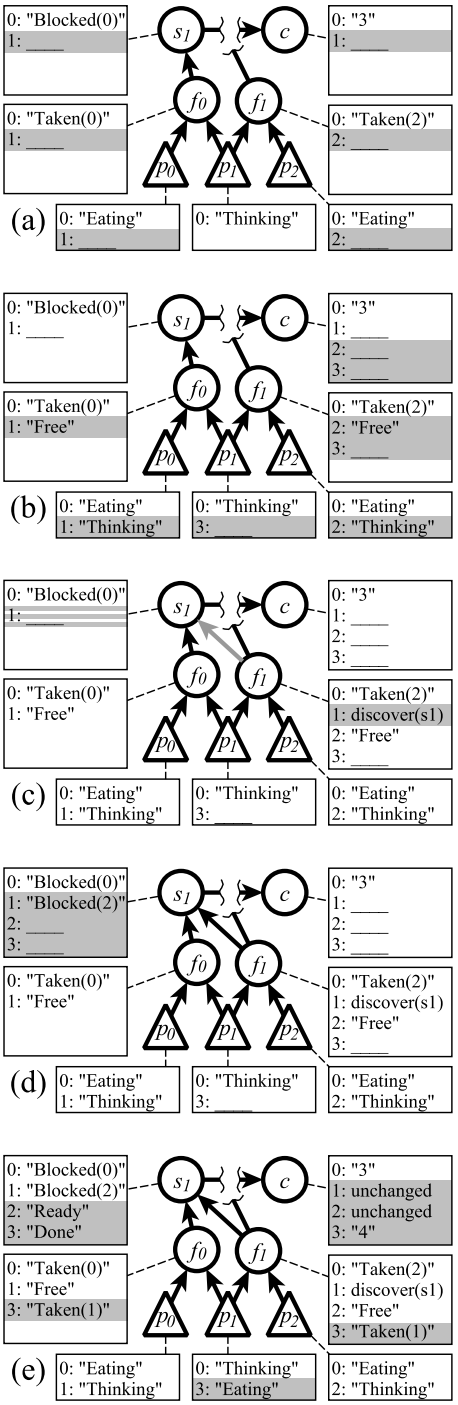


Fig. 8. MV-RP philosopher execution example.

condition requires t_1 to wait until all its predecessors (in the stored serialization graph) have also already switched to executing. This is necessary to prevent that executing transactions have predecessors that are still framing and thus may have not allocated all their potential reevaluations yet. Otherwise, an executing transaction could miss a predecessor's (planned) write when reading from a node, thereby making it illegal for the predecessor to execute that write later on and thus forcing an abort. Since t_1 doesn't have any predecessors in our example, this second condition is met too, and thus it can switch to executing.

Propagation with Pipelined Parallelism. Once a transaction switched from framing to executing, it transfers control to the propagation algorithm to issue reevaluations. An issued reevaluation is executed as soon as the transaction has exclusive access to the node, i.e., when its placeholder is the first placeholder on the node. When the reevaluation completes, that placeholder is *cleared*, i.e., replaced by a regular version that stores the node's new value. The transaction thereby implicitly relinquishes exclusive access to that node, which then becomes available for reevaluations by following transactions. This eager unlocking enables *pipeline parallelism*, where concurrent updates can make progress simultaneously even in mutually affected regions of the DG.

Consider Fig. 8 (b) for illustration. Again, new or changed versions since the previous step are highlighted with a grey background. t_2 has finished framing, which created a placeholder on c – this also established the order that t_1 precedes t_2. Despite their overlapping in the DG, both were allowed to start propagating their changes: t_1 has reevaluated p_0 and f_0, t_2 has reevaluated p_2 and f_1. Concurrently, transaction t_3 was created by a change of p_1 to `Eating` and has started framing. It has reached c over f_1 to create placeholders, but has not created a placeholder on f_0, yet.

Fig. 8 (c) shows how MVCC enables abort-free execution of reads and motivates the need for retrofitting. It depicts the application's state during reevaluation of s_1 by t_1; the placeholder is highlighted with a striped grey background. Because t_1 updated f_0 to `Free`, the user computation of s_1 now re-establishes its dependency on f_1 (grey dependency edge), which first reads its value. The latest value of f_1 is `Free` written by t_2. But, as t_1 precedes t_2, it is not supposed to see this value – the real time order in which t_2 wrote `Free` and t_1 reads f_1 does not match the serialization order. Without MVCC, the scheduler could not execute this read and would have to abort t_1. With MVCC, the scheduler can avoid the abort by returning a valid older value (`Taken(2)` from the initial state).

Note that there may be multiple older values that can be returned while upholding serializability. This happens if the reading transaction is not ordered against all existing versions' transactions, yet. In this case, a choice must be made how to establish the missing ordering relations, subject to two restrictions. (a) The read must be ordered before any framing transaction's placeholders, to maintain that executing transactions never have framing predecessors (cf. end of framing phase). (b) The read must be ordered after any already completed transactions, to ensure *strict* serializability. To follow real time order as closely as possible, MV-RP resolves such ambiguities by ordering transactions as late as possible under the restrictions of (a) and any already established relations, which automatically fulfils (b).

After the user-defined computation completes, s_1 must update its incoming dependency edges accordingly, i.e., it must add the grey edge to the outgoing dependencies of f_1. This write, however, suffers the same mismatch between real time and serialization order as the read did. This mismatch is visually apparent by the `discover(s1)` version (a third kind of version besides placeholders and written values, dedicated to dynamic edge changes) having been inserted in the middle of the history of s_1 (serialization order), instead of at the end (real-time order). Because t_1 precedes t_2 and t_3, the latter two must see this write. But, both have already read the outgoing dependencies of f_1 when the write by t_1 did not exist yet, and have therefore continued their execution based on the

value from the initial state. Thus, the scheduler can no longer execute the write by t_1 and would normally have to abort t_1. This is where retrofitting comes into play.

Retrofitting. To remain abort-free, the scheduler does allow t_1 to write the edge insertion, which temporarily invalidates serializability. To restore serializability, retrofitting then rewrites all version histories as necessary to reconstruct a state, where t_2 and t_3 read the outgoing dependencies of f_1 only after t_1 added the edge $f_1 \longrightarrow s_1$. Fig. 8 (d) shows the changes made by retrofitting (lines with grey background). Retrofitting computes the required corrections from the version history of the source node of the added edge (here, f_1).

(1) For every transaction that has a written or placeholder version that is ordered after the `discover(s1)` version (here the `Free` version of t_2 and the placeholder of t_3), retrofitting creates placeholders on all nodes reachable from the edge's sink (here all nodes on the path from s_1 to c). The history of s_1 in Fig. 8 (d) has two new placeholders labelled 2 and 3 (c already had placeholders for t_2 and t_3). This compensates for the fact that the framing phase of t_2 and t_3 originally missed the edge that is being added ($f_1 \longrightarrow s_1$).
(2) For every written version after the `discover(s1)` version (here version `Free` of t_2), retrofitting additionally instructs the propagation algorithm of the respective transaction (t_2 in this case) to enqueue the edge's sink node (s_1) for reevaluation. This compensates for the fact that the change propagation of t_2 originally missed the edge being added ($f_1 \longrightarrow s_1$).

For dependency edge removals, both compensations are reversed: Placeholders are removed and obsolete reevaluations are deleted from the propagation algorithm's queue.

By changing the reevaluation queue, the scheduler interferes with the reevaluation order devised by the propagation algorithm. As the original order guarantees glitch freedom, the retrofitting changes must retain this guarantee, which we prove: A dependency edge change always occurs during the reevaluation of the edge's sink node. This means, the reevaluating transaction has the first placeholder on the sink node and any nodes reachable from there. Because the sink node's reevaluation is still ongoing, the sink node is not glitch-free yet, and thus none of these placeholders can execute their reevaluations and be cleared. Thus, no subsequent transactions can start or have started a reevaluation on any of these nodes, because their placeholders cannot be the first on any of these nodes. Hence, no values can exist yet on any of these nodes, which could have been glitch-free before, but due to the retrofitting changes now no longer are.

The state in Fig. 8 (d) corresponds to one where t_2 and t_3 started after t_1 added the edge $f_1 \longrightarrow s_1$. This shows that retrofitting successfully rewrote all affected nodes' version histories into a correct result, despite the timing mismatch of the dependency discovery. From this point on, t_3 can complete framing and all transactions can complete their change propagation normally. The ultimate result, shown in Fig. 8 (e), is (view-)equivalent to a serial execution of t_1 before t_2 before t_3.

4.3 Properties of MV-RP

Theorem 1 (Abort-Free Strict Serializability)**.** If all mutable state, used by any reactive's user computation to compute its result, is implemented as other reactives and always accessed through `before`, `after` and `depend` calls, then MV-RP produces only abort-free strict serializable histories.

Theorem 1 states, that MV-RP fulfils our correctness definition from §3.3. Note that the added premise does not represent a practical hindrance for its applicability. The premise only limits the theorem's applicability under user computations that essentially circumvent the RP abstractions by reading some non-constant parameter values outside of the scheduler's control.

For space reasons, we provide only an intuition of the theorem's proof here; the full proof is available in an extended technical report [Drechsler et al. 2018]. Our proof is built upon a pseudocode implementation of the scheduling routines and established multiversion theory. Multiversion

theory defines rules to construct a *multiversion serialization graph* from a history of individual reads and writes. The multiversion theorem, proven in literature [Bernstein et al. 1986], states that a given history is serializable if its multiversion serialization graph is acyclic. Recall that the MV-RP scheduler tracks the *stored* serialization graph across the composite RP operations. The stored serialization graph is intended to represent the multiversion serialization graph, but both are different graphs. The core of our proof revolves around showing that this representation is truthful. Our proof first builds several supporting lemmas. It shows, that reevaluations are correctly executed sequentially on each node, in sync with the serialization order. It then shows, that all user-visible read operations (i.e. those on nodes' values v_n) correctly await all preceding transaction's placeholders to be cleared. Lastly, it shows that for each individual read and write, once it is executed by the pseudocode routines, a node version exists on some node, such that for all edges that the individual read or write adds to the *multiversion* serialization graph, an equivalent path exists in the *stored* serialization graph. In other words, the scheduler's stored serialization graph is an over-estimation of the actual multiversion serialization graph, containing at least all its paths or possibly more. This proves that the multiversion serialization graph is acyclic, since any cyclic path would also exist in the stored serialization graph, but the latter is grown such that a cycle is never introduced, so no such path can exist. This in turn means, that MV-RP always fulfils the premise of the multiversion theorem, which thus implies that all histories that MV-RP can produce are serializable. As we already discussed above, the extension to *strict* serializability is achieved by choosing always the latest possible value to return for reads. Lastly, the absence of aborts is trivial, since aborts do not occur naturally and the scheduler does not implement any routines that introduce aborts for transactions.

A noteworthy property of MV-RP beyond its correctness is that its integration into the RP architecture leaves the user-facing API unchanged. That is, the integration is "syntactically transparent" in the sense that no syntactic adaptations of user programs are needed. This is visually apparent in that the user-defined half in Fig. 3 and Fig. 7 is identical, including all operation arrows across the dashed border into the RP-internal half (which represent this API). Further, since the scheduler is abort-free, the integration is also "semantically transparent" in the sense that its effects are not observable in the application behavior. As a result, existing REScala code not written with multi-threading in mind, e.g., the code of the philosopher example shown in §2, can run without any changes on top of MV-RP ensuring its correct execution in a multi-threaded environment.

Notably, the transparent and correct multi-threaded execution includes support for dynamic dependency changes. This means that MV-RP supports the full expressiveness of dynamically reconfigurable applications. E.g., it is possible to grow and shrink applications' dependency graphs by individual nodes[4], or merge and split entire applications' graphs. This is possible even at run-time, while other updates are propagating concurrently, and without such changes having been foreseen or planned during development of the applications. Any number of edges can be added or removed as part of a single transaction, which yields consistent semantics for atomically (dis-)connecting new or obsolete nodes or subgraphs. Retrofitting automatically (de-)schedules superfluous or missing reevaluations of newly (dis-)connected nodes as needed.

4.4 Explicit Transactions

By default, any imperative interaction (any call to `s.now` or to `update(`$\texttt{i}_1$ `->` $\texttt{v}_1$`,` $\texttt{i}_2$ `->` $\texttt{v}_2$`, ...)`) is executed as one transaction. In some cases though, one may need an atomic unit of work that consists of multiple imperative interactions. E.g., consider a UI button that fires its `clicked` Event only if

[4]In fact, this is what happens when a RP application is started. When code such as that from §2 is executed, it imperatively instructs new nodes to be created one by one. MV-RP then executes one transaction for each node, where the node is first instantiated and then connected to previously created nodes by dynamically inserting all required edges.

its `enabled` Signal reads true. Both the `enabled.now` read and `clicked.fire()` change propagation must be part of the same transaction. Otherwise, a second change propagation might disable the button concurrently, resulting in the button appearing disabled in its own click propagation transaction. For this purpose, MV-RP supports the optional feature of *extended transactions*.

The first form of extended transactions are *multiple read* transactions that consist of multiple `s.now` operations. As an example, consider the following code based on the philosophers example:

```
2   if (transaction{ fork(0).now == Free && fork(1).now == Free }) phils(1).set(Eating)
```

Because the transaction scope ensures that both forks' reads are executed as one atomic unit, this code behaves identically to a singular `sights(1).now == Ready` read like in Fig. 1, Line 3.

The second form of extended transactions are *read-update-read* transactions, which may contain a single `update(...)` call surrounded by any number of `s.now` reads. These transactions allow to express updates with atomic pre- and post-conditions. They come with one complication related to the fact that the scheduler must know the set of possibly changed inputs before starting to execute the transaction. For a single `update(`i_1 `->` v_1`,` i_2 `->` v_2`, ...)` call, MV-RP automatically infers $\{i_1, i_2, ...\}$ from the call's parameters. But for a call to `transaction{...}` with an arbitrary user-defined closure as the only parameter, this is not feasible. Instead, MV-RP requires this set to be manually declared as a second parameter to the `transaction` scope. With read-update-read transactions, we can finally implement the loop iteration from Fig. 1 such that forks are never used by two philosophers at the same time, by grouping the `sights(idx) == Ready` check and the `phils(idx).set(Eating)` update into a single transaction:

```
2   transaction(Set(phils(idx))) { if(sights(idx).now == Ready) phils(idx).set(Eating) }
3   if(sights(idx).now == Done) phils(idx).set(Thinking)
```

4.5 Garbage Collection

Any transaction is considered *complete* only once it executed all its operations *and* all preceding transactions in the stored serialization graph have already completed, too. This restriction mirrors that of the transition from framing to executing, but for a different reason. Even if a transaction has no more active or queued reevaluations, its predecessor transactions may perform dynamic edge changes which necessitate retrofitting and thus enqueue further reevaluations. The only way to be sure that no further reevaluations will occur is thus to wait until all predecessor transactions have completed first. Transactions are deleted from the stored serialization graph once they are complete.

Node versions must be retained as long as any later transactions might still read them. A written version of a completed transaction in some node's history is a sufficient condition for all *older* versions in that history to be deleted. Versions can be deleted eagerly, by each transactions upon completion traversing all nodes where it created a version and deleting all older versions. Alternatively, old versions can be garbage collected in batches periodically, in some way that minimalizes computational overhead.

5 EVALUATION

We evaluate the performance of MV-RP relative to several factors: relation between the cost of user computations and the synchronization overhead, contention level, DG topology, and cost of dynamic edge changes and handling of their conflicts (retrofitting). In §5.1, we evaluate, which topologies MV-RP can parallelize, and how costly user computations must be for this to be efficient. The experiments in §5.2 investigate scalability as a function of overhead, which itself is a function of contention, in benchmarks with very cheap user computations, i.e., where scheduling overhead dominates execution costs. §5.3 analyzes the performance costs related to dynamic edge changes. Lastly, §5.4

quantifies, how much effort is required to parallelize the *existing* "Universe" RP application from REScala's example corpus, and the effect this has on the application's performance.

Another desirable evaluation approach involves measuring the effect of using MV-RP in real complex applications beyond the Universe example. But, even evaluating the effects of MV-RP on multiple complex applications yields no guarantees that the results generalize to other/future applications. Yet more critically, since MV-RP is the first system to enable serializable multi-threaded RP, no such applications exist yet. Thus, we would have to build one (or more) ourselves for the sole purpose of our own evaluation, which would unavoidably introduce significant bias. Therefore, our evaluation does not investigate this direction, but focusses only on building a set of rules that do allow users to generally estimate the impact that given features of any applications have on their performance, through the above-mentioned microbenchmarks approach.

Our evaluation compares MV-RP with several alternative scheduling approaches:

- G-Lock implements global locking by wrapping each transaction execution in a global `synchronized{..}` scope. This prohibits concurrent executions, but thereby trivially fulfils our correctness for all applications.
- Handcrafted refers to application-tailored manual locking implementations, which we include whenever feasible. In Handcrafted implementations, no synchronization is integrated into the RP runtime. Instead, the imperative user code is changed to manually acquire and release locks before and after executing any `now` or `update(...)`. Handcrafted solutions have little overhead, but often require significant effort to be devised individually for each application, and are fragile in that they do not generalize to other applications and easily break when parts of the application are changed. Note that more elaborate Handcrafted solutions become possible when also acquiring and releasing locks inside of reactives' user computations (this would, e.g., enable some cases of pipeline parallelism). But, such solutions are much more complex to explain, and extremify above-mentioned negative aspects so much that we consider them unreasonable and hence do not include them in the evaluation.
- STM-RP is an integrated RP scheduler that stores all node variables in ScalaSTM [Bronson et al. 2010], a library-based off-the-shelf software-transactional memory. Each transaction execution is wrapped in an `atomic{...}` scope and executed optimistically, aborting and restarting upon concurrency conflicts. STM-RP thus does not fulfil our correctness: it provides strict serializability, but is not abort-free. It is therefore applicable only to applications without side-effects; for compatibility, all our benchmarks adhere to this.

In all experiments, we measure the throughput (more is better) for an increasing number of threads that concurrently execute updates. All experiments run on computers with dual Intel Xeon E5-2670 CPUs for 2x8 cores at 2.6 - 3.3 GHz, using the OpenJDK JMH benchmark harness in the 64-Bit Oracle JRE 8u141 at 1 GB fixed heap under CentOS Linux 7.4. In general, shown results are average measurements of executing each benchmark operation in every configuration for at least 35 seconds (after at least 25 seconds unmeasured warmup), repeated six times on fresh JVM instances each. Every single data point in each of the following chart thus represents at least 3.5 minutes of execution; a data point of, e.g., 50 ops/ms is the average measurement based on the benchmark operation having executed over 10.5 million times.

5.1 Cost of User Computations and Impact of Topology

Our first set of experiments serves a twofold purpose. First, they analyze, how well MV-RP can parallelize concurrent updates on different topologies. Second, we analyze, how costly user computations have to be so that scheduling overhead doesn't reduce throughput more than scaling increases it. For the first goal, the experiments use a set of minimalized fundamental topology building blocks. We analyze scalability by measuring throughput across 1 to 16 threads concurrently

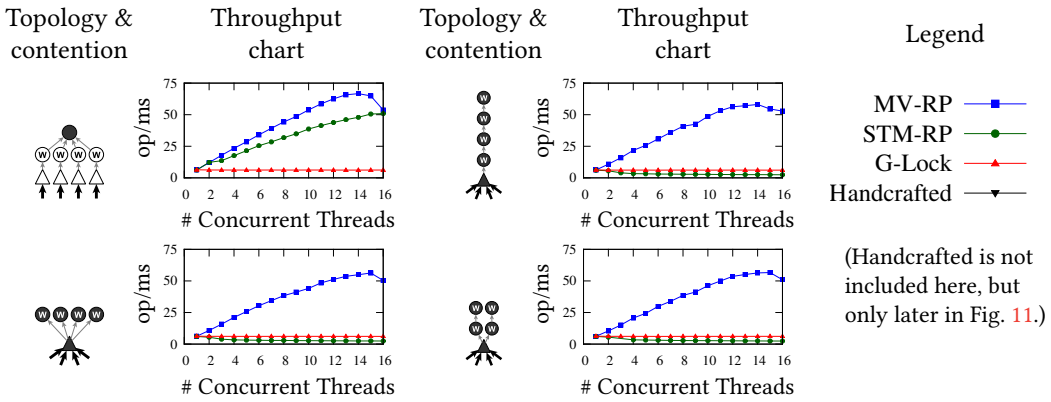


Fig. 9. Scalability across base topologies of updates with approx. 160 µs user computation time.

executing updates on each topology. We sized each of these topologies to contain precisely 16 designated work nodes, so that maximum scalability can be achieved, but only if the framework actually manages to parallelize each topology down to single nodes. For the second goal, through trials on each topology, we pinpointed the minimum amount of work per work node, such that MV-RP provides scalability up to just below 16 threads. We choose the target point just below 16 threads so that the corresponding pinnacle point in the throughput graphs is visible and shows that the selected amount of work is precisely as large as necessary.

Fig. 9 shows the topologies and resulting throughput graphs. In the topology visualizations, thicker arrows pointing towards input nodes represent active threads admitting changes. The color of nodes visualizes a heat map of how much contention these corresponding updates cause on each node, with darker shades representing higher contention. In the top left experiment, all updates perform some uncontended workload before funneling into a maximally contended global bottleneck node. For this topology, each update affects only a single work node (marked W). Scalability to just below 16 threads (shown in the throughput graph) is reached at a computational cost (on the hardware we used) of only ca. 160 µs user computation time per work node.

Notice that all topologies in this set of experiments have at least one single node bottleneck for all updates. Achieving scalability thus requires support for pipeline parallelism. MV-RP features pipeline parallelism by design. Handcrafted solutions cannot provide pipeline parallelism (without managing locks from inside user computations), so no Handcrafted measurements are included. STM-RP can support pipeline parallelism, but only under lucky external circumstances. Such circumstances are given in this first topology (conflicts can only occur for a brief moment at the very end of each transaction), but not in the others, thus STM-RP provides scalability only for this first topology.

In the bottom left experiment, the top left topology is reversed so that updates first pass the bottleneck and only afterwards reach the parallelizable work nodes. This way, all nodes in the graph are contended by all threads. To parallelize this topology, multiversion concurrency control is indispensable: to concurrently recompute all work nodes in different threads, all threads written values on the topology's source node must be available to be read simultaneously. The scalability shown in this experiment's throughput graph was achieved with each work node performing only ca. 10 µs of computations, which adds up to again ca. 160 µs for each entire update.

In the top right topology, the work nodes are arranged in a "chain" topology, and the bottom right "grid" topology is a combination of the previous two experiments, with work nodes arranged in 4 chains of 4 nodes length each. The throughput scalability shown for both the chain and grid topologies was also achieved with ca. 10 µs of computations per node, i.e., ca. 160 µs per update.

All topologies in this set of experiments thus have their inflection point at just below 16 threads at the same amount of computations per update. The reason for this is as follows. The inflection in these charts occurs, when the capability of the scheduler to order transactions reaches its limit and becomes a bottleneck for the transaction throughput. This bottleneck *is* dependent on the topology, but only indirectly, in that it depends on how often transactions need to be ordered in which numbers, which is a consequence of the DG topology. It is independent though, of how the cost of reevaluations is distributed across the topology. The topologies in this set of experiments all have the same inflection point because they differ only in the second aspect, but not in the first (in each topology, every transaction conflicts with all others upon reaching the bottleneck node). We evaluate the effect of differences in the first aspect in the next set of experiments.

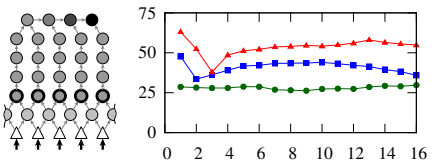
Summarizing this set of experiments, we conclude that MV-RP with multiversions and pipeline parallelism can parallelize RP updates down to individual nodes, regardless of the DG topology. The necessary computational cost of user computations per update to overcome the overhead cost of scheduling up to almost 16 threads is only ca. 160 µs (on our hardware, and assuming that work is spread evenly across at least 16 nodes). This cost can mostly be considered an upper bound, because – as the following experiments will also show – the bottleneck becomes much less restrictive when there is less contention between concurrent updates.

5.2 Very Cheap User Computations

Next we analyze scheduling overhead and its impact on scalability in more detail. To do so efficiently, we use an application with cheap and fast updates, so that the scheduling overhead dominates the cost of executions and is therefore heavily emphasized in the measurements. Our philosophers application from §2 serves well for this, since the majority of its nodes executes only a single integer addition and equality check, or even less. The operation (op) that we use as a benchmark always performs two updates: First, Line 2 from the end of §4.4 is repeated until the if-condition passes and the update to `Eating` is executed once. Afterwards, a second update changes the philosopher back to `Thinking`.

Three aspects of MV-RP cause overhead: the framing phase, operations on nodes must search for their correct placement in the version histories, and transaction relations must be recorded in the stored serialization graph. Clearly, these aspects easily dominate the node computations of the philosopher application in execution cost. Moreover, the latter two aspects are more expensive under higher contention (version histories contain more elements and more ordering relations between transactions are created). Searching node version histories doesn't necessarily hinder scalability since it is parallelized along with the node operations (version histories of different nodes can be searched and changed concurrently), but this also conflicts more frequently under higher contention. Contention is therefore the most influential factor on scheduling overhead, and thus also on scalability if updates are cheap. We thus run different configurations of philosophers that produce different amounts of contention, to evaluate this entire space.

Extreme Contention. Fig. 10 on the left shows the topology of the philosophers application from §2. This original configuration is a fairly extreme worst-case in terms of contention. All `forks` are contended by two threads, all `sights`, Events, and the folding Signal `counts` by three, and the summing-up Signal chain (`totalCount`) successively funnels all 16 threads into a single node bottleneck. This bottleneck again means, we cannot devise a Handcrafted solution better than global locking (without managing locks from inside user computations). Thus, we compare only G-Lock, MV-RP and STM-RP.



Axes' units and lines' labels as in Fig. 9.

Fig. 10. Extreme contention.

The chart on the right of Fig. 10 shows the throughput graph of 1 to 16 threads running on a table of 16 philosophers.[5] First, consider the data points for one thread. The throughput of G-Lock represents synchronization-free single-threaded performance of RP updates, as the overhead of acquiring an uncontended global lock before each change propagation is negligible. With about 75 op/ms (recall, each op is two updates plus two `.now` reads in single-threaded measurements), this corresponds to each update executing in about 6.5 µs time, i.e., updates are indeed very cheap. Hence, the overhead cost of the other schedulers in a single-threaded environment stands out clearly: application performance drops by 25% with MV-RP, and by 55% with STM-RP.

Next, consider the configuration with two threads. G-Lock still executes the same workload, but has become noticeably slower because the global lock is now contended by multiple threads, i.e., the CPU has to engage its synchronization mechanisms, which makes the lock acquisition more expensive. For MV-RP, this situation is more complex. With pipeline parallelism, MV-RP allows very fine-grained parallelization of RP applications (shown in detail in §5.1). With only two threads executing on 16 philosophers, contention is still low enough that they can execute most of their workload (including scheduling overhead) concurrently. Nevertheless, MV-RP also becomes noticeably slower, i.e., its scheduling overhead increased significantly. Aside from also having some contention on the nodes' locks, this is mainly caused by its initially expensive serializability mechanisms (large histories, transaction ordering) now having to engage.

For three and more threads, the throughput of G-Lock and STM-RP remains constant, with STM-RP around 50% lower than G-Lock. For G-Lock this is expected (it prohibits concurrency), for STM-RP this is because it cannot parallelize updates under high contention of the application. MV-RP is successful in parallelizing the application's updates, indicated by its throughput initially increasing up to around 9 threads. This shows that while initially engaging its serializability mechanisms significantly increased its overhead, adding on more threads afterwards has less of an impact. Beyond 9 threads, the potential for parallelization becomes increasingly saturated while the overhead continues to grow, and thus the throughput decreases again.

Overall, in a setting with extreme contention and very cheap updates the performance of MV-RP remains between 5% and 30% lower than G-Lock, i.e., MV-RP fails to improve performance despite successfully parallelizing updates. This is to be expected because in this setting the synchronization overhead clearly dominates the cost of what can be parallelized. On the other hand, the overhead does not make MV-RP unreasonable to use. We consider this scenario an example at the lower bound for the usability of MV-RP. In the following experiments in this subsection, we show that lowering contention allows MV-RP to succeed in scaling performance, even if updates remain cheap and synchronization overhead still dominates the cost of each update's execution.

High Contention. First, by removing the summing-up signal chain (i.e., remove Line 29 in §2), we remove the most extremely contended nodes including the bottleneck from the application. The remaining topology still induces a high amount of contention with most nodes still being contended by three threads, but no longer contains nodes contended by more than three threads. Its contention heat map is labelled "High" in the first row, left-most column of Fig. 11.

Without the bottleneck, we can devise a Handcrafted synchronization: For each `phils(i)`, a `java.util.concurrent.locks.ReentrantLock` referenced as `lock(i)` is added. In order to update a given `phils(i)`, the executing thread must hold the philosopher's own `lock(i)`, and both neighbors' `lock(i-1)` and `lock(i+1)` (out-of-bounds indices wrap around). This way, the only interaction that can occur between concurrently executing updates are concurrent reads affecting some shared

[5] When there are more philosophers than threads, philosophers are distributed to threads round-robin and each thread for each measured op picks a random philosopher from its assigned pool (e.g., thread #2 of 5 randomly chooses `phils(2)`, `phils(7)` or `phils(12)`).

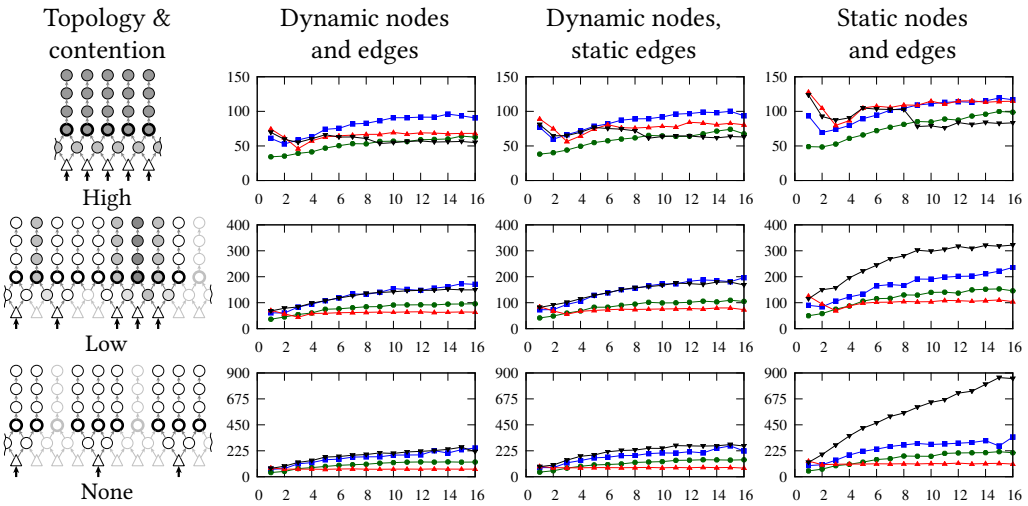


Fig. 11. Throughput for different configurations of the philosophers application.

`forks`, which is harmless since reads are idempotent and commutative. Threads simply block to acquire needed locks, ordered by their indices to prevent deadlocks. E.g., to update `phils(0)`, locks `lock(0), lock(1), lock(15)` are acquired in that order. Similar to G-Lock, this Handcrafted approach has almost no overhead (three locks per update, instead of one), but permits parallelization.

The left-most chart in the first row (second column) of Fig. 11 shows throughputs again on a table of 16 philosophers. Without the summing-up chain, throughput of G-Lock has increased by about 20%, but follows the same shape as in Fig. 10. Handcrafted performs similar to G-Lock and does not achieve any scalability. This is because threads queue up while blocking to acquire their needed locks, and then transitively waiting even for threads with non-interacting updates[6]. This shows that such simplistic locking is unsuitable for scenarios with high contention. Adding more sophisticated mechanisms for dealing with such issues, however, is usually not reasonable, as it would require to invest yet more effort into a fragile synchronization that cannot be applied to other topologies and breaks under changes.

For MV-RP, removing the summing-up-chain reduced single-threaded overhead to 20%, for STM-RP it is still at 55%. Further, despite the still very high amount of contention causing significant overhead, MV-RP already does manage to scale throughput, surpassing the performance of G-Lock and Handcrafted from three threads onwards. This is due to pipeline parallelism providing more fine-grained mutual exclusion than Handcrafted, blocking threads only on individual nodes and only as long as necessary, thus threads do not queue up needlessly. STM-RP also achieves speed-ups, but does not manage to outperform G-Lock even at 16 threads.

Low Contention. To investigate low contention scenarios, we spread threads more thinly by using a larger table of 64 philosophers. The corresponding topology, with a contention heat map for an exemplary thread distribution snapshot, is labelled "Low" in Fig. 11 (second row, left-most column). The left-most chart in that row (second column) shows the experiment's results. G-Lock is the same as under high contention, which is expected since the workload per thread is the same. Contention is low enough that Handcrafted no longer forms excessive queues and can scale across

[6]The sudden drop in throughput from 8 to 9 threads, which becomes more exacerbated in later experiments, is caused by queueing times becoming long enough to cross the threshold where `ReentrantLock` moves from spinning until the lock becomes available to de-scheduling the requesting thread instead, which leads to a sudden increase in overhead.

all 16 threads. The throughput of MV-RP scales about twice as well as under high contention for large thread counts. Even under low contention, MV-RP still scales better than Handcrafted, from its 20% lower single-threaded throughput to outperforming Handcrafted by about 10% at 16 threads. STM-RP also manages to scale above G-Lock from three threads onwards, but still achieves only approximately 60% of the performance of MV-RP and Handcrafted.

No Contention. As a final variation, we use a fixed placement of one thread on every fourth philosopher. This allows all threads to execute concurrently without their updates ever interacting, resulting in zero contention. The corresponding topology is labelled "None" in the last row of Fig. 11. Nodes not reevaluated by any thread are faded out through grey outlines, to visualize the conflict avoidance. The corresponding throughput graph is shown next to it (third row, second column). G-Lock is again as before, but all other schedulers can now scale freely, without their differing concurrency management approaches having any effect. Their relative differences are thus equal across all thread counts (modulo some measurement noise): MV-RP 20% below Handcrafted, STM-RP 55%.

Comparing this contention-free spacing to the previous experiments, we can now estimate the cost of contention for each scheduling approach on this application. At 16 threads, low contention costs about 35% performance when using Handcrafted, 25% when using MV-RP, and 15% when using STM-RP. High contention respectively costs about 75% performance under Handcrafted, 60% under MV-RP, and 45% under STM-RP. Extreme contention is not comparable due to its different workload (the removed summing-up chain).

Summary. To recap this set of experiments, we have shown that even for updates of only few computations under high contention, MV-RP is capable of providing scalability with multi-threading. STM-RP is least affected by contention, but has significantly more overhead, resulting in performance consistently and significantly lower than MV-RP and Handcrafted, in addition to its weaker guarantees (not abort-free). MV-RP manages to outperform Handcrafted under low contention and even more so under high contention (despite higher single-threaded overhead). Considering in addition that MV-RP is usable out-of-the-box and applicable to all topologies, whereas Handcrafted must be manually developed and maintained for each application, MV-RP is clearly the best choice here.

5.3 Cost of Dynamic Dependency Changes and Retrofitting

To evaluate the scheduling overhead for executing dynamic dependency changes, as well as for handling the conflicts they produce (i.e., retrofitting), we modify the behavior of nodes that perform dynamic dependency changes in the philosophers application, i.e., all `sights` nodes. For visualization, these nodes are highlighted through bold outlines in the topology overviews of Fig. 11. They account for ca. 23% of executed reevaluations, with close to 10% of all reevaluations actually executing a dynamic dependency change.

First, we changed the behavior of `sights` such that they always access both forks they depend on, but are still declared as dynamic nodes. The results for these experiments are shown in the second-to-right column of Fig. 11, labelled "dynamic nodes, static edges". This way, no dynamic dependency changes occur, but the remaining workload of reevaluating these nodes stays nearly the same in that the RP framework still must collect, which dependencies were accessed, and check if any changes occurred compared to the previous reevaluation. For the high and low contention configurations (first and second row), this removes both dynamic dependency changes and their according retrofitting. For the contention-free configuration (third row), the lack of interaction between concurrent updates means that the previously executed dynamic dependency changes never required any retrofitting, and thus only dynamic dependency changes but no retrofitting

was removed in this configuration. Considering the contention-free configurations first, we see that all scheduling approaches show an insignificant increase in performance across all threads. In particular, this includes G-Lock, which implies that this change cannot be attributed to any operation-specific scheduling overhead, as G-Lock has no such thing. From this we conclude that dynamic dependency changes have no noticeable cost overhead in any of the scheduling approaches. Considering the low and high contention configurations, we observe the same changes, meaning the handling of conflicts of dynamic dependency changes (i.e., retrofitting in case of MV-RP) on top of executing the changes themselves also has no noticeable impact on performance.

Concluding the philosophers experiments, we consider a final variation of the `sights` nodes, where they declare both forks as their static set of dependencies at initialization, and the RP framework thus no longer collects and compares their accessed dependencies during each reevaluation. This removes a significant chunk of their reevaluation costs, but this chunk consists only of thread-local computations that do not involve the schedulers, i.e., scheduling overhead remains unchanged. The results are shown in the right-most column of Fig. 11, labelled "static nodes and edges". As expected, with less computations to be executed under the same scheduling overhead, we observe noticeable increases in throughput compared to the previous set of experiments (second-to-right column) across all scheduling approaches, but the schedulers' overhead is emphasized stronger, placing them relatively further apart. In particular in the contention-free setup, the throughput of Handcrafted is much higher than MV-RP or STM-RP, because it has the least overhead (just three locks per update). Comparing this to the low contention scenario though, shows that this advantage quickly diminishes, and at high contention even disappears entirely.

Summary. To recap the results from this set of experiments, both the cost of scheduling for dynamic dependency changes and of handling the conflicts they cause (i.e., retrofitting) are negligible, in particular compared to the significant cost of using dynamic dependencies in the first place.

5.4 Parallelizing Existing Applications

As a final experiment, we took an existing REScala case study called "Universe" to evaluate two research questions: First, how much effort and code changes are necessary to migrate an existing single-threaded RP application into a multi-threaded environment safely. Second, can this migration improve the application's performance. We chose "Universe" for this because it is one of the largest applications in REScala's example corpus, and because it has a structure that makes it easy to introduce multi-threading in its imperative parts.

For the first question, we found that no changes to the RP code of the application were necessary at all. The only modifications we made were to the application's imperative code, and these served only to introduce multi-threading, not to facilitate the integration of MV-RP: "Universe" implements a simulation of an ecosystem with plants and animals, which repeatedly executes two phases. First, a list of tasks for animals and plants to move, feed, grow etc. is populated through a single RP update (i.e., this phase is not parallelizable). Second, all these tasks – each of which is again a RP update – are executed. Here, we made a single change in that we use a parallel drop-in replacement from the standard Scala collections library for the list of tasks, which submits all tasks to a thread pool for concurrent execution instead of running them one by one.

For the second question, Fig. 12 shows the throughput under up to 16 worker threads. Since the application is only a simulation, it does not use side-effects, and we were able to include STM-RP in this experiment. We do not include Handcrafted results, as we deemed too large the effort of not only precisely analyzing and understanding the possible topologies of the application's dependency graph, but then also trying to find, how – or if at all – manual synchronization

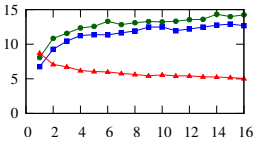


Fig. 12. Universe case study.

is possible. As a rough intuition though, the application executes a large number of updates across a very wide graph, meaning even at 16 threads there is still very little contention during each second phase. Hence, STM-RP is equally capable as MV-RP in scaling the application's throughput through multi-threading. Throughput increases less quickly under higher worker threads because only each iteration's second phase is parallelized, but first phase execution times remain constant. Further noteworthy is the application's unusually heavy use of node creations and removals when animals and plants are spawned and killed. This gives STM-RP a slight advantage over MV-RP, because the data structures of STM-RP on the nodes have a smaller memory footprint than those of MV-RP, and are thus faster to instantiate.

In conclusion, we find positive answers for both research questions. The necessary effort for migrating to safe multi-threaded execution is minuscule, and performance increased noticeably.

5.5 Summary

In summary, we draw the following conclusions. First, we have found that the topology of the dependency graph and whether or not the application uses and possibly has conflicts from dynamic edge changes do not affect the scalability provided by MV-RP. The only two dimensions we found to have an impact are conflict density, which depends on how many threads are spread how densely across a given topology, and the cost of user computations during reevaluations. Fig. 13 visualizes these dimensions and intuitively positions the experiments we conducted in this space. This shows that MV-RP will be beneficial to use for most applications except those that execute updates of only very cheap computations under extreme thread contention. However, even for those worst-case scenarios MV-RP is feasible to use with only some overhead over global locking. It is also worth noting that MV-RP is still a very young implementation with several potential optimizations not yet explored or even discovered, so it still has potential for future performance improvements.

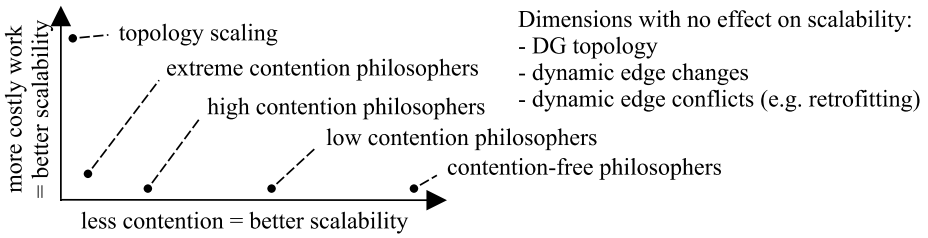


Fig. 13. Effects of dimensions of RP applications on MV-RP scalability.

6 RELATED WORK

(Functional) Reactive Programming [Elliott and Hudak 1997] explores time-changing values, and advanced type systems have been developed that guarantee bounded-space execution [Krishnaswami et al. 2012], space and time leaks freedom [Krishnaswami 2013] and liveness [Jeffrey 2013]. [Ramson and Hirschfeld 2017] investigates which fundamental abstraction can be used to implement different RP languages. [Kamina and Aotani 2018] proposes a core RP calculus solely based on Signals. Another line of research embeds RP abstractions into existing languages. FrTime [Cooper and Krishnamurthi 2006] is a RP in Scheme. Scala.React [Maier and Odersky 2013] and REScala [Salvaneschi et al. 2014b] implement RP for Scala. Flapjax [Meyerovich et al. 2009] extends Javascript. Many Javascript frameworks include concepts of RP, e.g., Angular.js, React.js, Bacon.js, Knockout, Meteor and Reactive.coffee. To our knowledge, none of these frameworks support concurrent updates.

Parallel FRP [Peterson et al. 2000] duplicates bottleneck event streams in the DG to increase throughput, but multiple updates still cannot progress concurrently. Reactive Concurrent Programming [Amadio et al. 2006], despite its very similar name and also providing an abstraction called signals, is unrelated to FRP and RP signals; it is instead a programming language for collaborative multi-threading, which by "signals" refers to an abstraction that imitates traffic lights, where concurrent threads temporarily stop and wait for each other to synchronize their progress.

Lastly, there are several (F)RP systems that address concurrency and support parallelism in different ways; we discussed their properties and relations to our work in §1.

Distributed RP systems are intrinsically subject to concurrency. AmbientTalk/R [Lombide Carreton et al. 2010] is RP for mobile peer-to-peer networks. It supports loosely coupled networks instead of providing distributed strong consistency like glitch-freedom, and local propagations on each host are single-threaded only. Distributed REScala [Drechsler et al. 2014] implements distributed RP with glitch-freedom, but uses global locking to avoid concurrency problems. DREAM is a RP middleware [Margara and Salvaneschi 2014, 2018], supporting concurrent updates with choosable consistency levels, including glitch-free and transactional, but it does not support dynamic dependency changes. Fault-tolerant Distributed RP [Mogk et al. 2018] introduces error propagation and crash recovery for reactives and executes updates glitch-free by mutual exclusion on each host, but supports distribution only through conflict-free replicated data types with eventual consistency and therefore supports some concurrency in the global propagation, but only with eventual consistency instead of glitch freedom. (F)RP has been also studied in combination with multitier programming for distributed applications in ScalaLoci [Weisenburger et al. 2018] and tailored towards web applications [Reynders et al. 2014]; these works allow reactives of arbitrary types to be sent to different hosts, but focus on the language design benefits and synergies of multitier and reactive programming instead of solving consistent distributed propagation, and therefore also only execute updates glitch-free by mutual exclusion per host, but do not provide glitch freedom across the whole application.

Incremental computing describes approaches that take existing concurrent algorithms and automatically incrementalize them. Different approaches address different programming models, parallelism and memory models, including functional fork/join-parallel programs [Hammer et al. 2007], imperative fork/join-parallel programs [Burckhardt et al. 2011], MapReduce programs for Big Data [Bhatotia et al. 2011], and even shared-memory multi-threaded programs [Bhatotia et al. 2015]. They all achieve incrementalization by recording a dependency graph of chunks of the original algorithm during an initial, non-incremental execution. Afterwards they incrementally update the result by selectively propagating input changes across this dynamic dependency graph, re-executing chunks as needed. These works therefore have significant similarities with RP change propagation, but they address a different dimension of parallelism. They incrementalize programs that already correctly use parallelism, and then replay this parallelism during singular incremental update propagations. Our work on the other hand addresses safely adding parallelism where there was none before, in the form of allowing multiple different incremental update propagations to execute concurrently.

Event Processing Languages and Middleware propagate events to interested clients, similar to change propagation in RP between Events. Publish-subscribe systems [Eugster et al. 2003] aim to support loose coupling among event publishers and observers. By contrast, complex event processing (CEP) [Cugola and Margara 2012] is about correlating events. Both rely on callbacks, which RP aims to avoid [Meyerovich et al. 2009]. Reactive extensions first became popular through Rx.NET [Liberty and Betts 2011], but are available in many languages today (e.g., RxScala – not to be confused with Scala.Rx). They provide convenient syntactic features for event streams and asynchronous computations, but no support for glitch-freedom. Stream processing is increasingly

supported in a number of languages, including Java 8 (the stream-based collections interface) and Akka streams, but is aimed at supporting pipeline and/or data parallelism of sequential or independent events, not at providing consistent semantics for interacting concurrent events.

Imperative concurrency control. Small-scale concurrency tools, e.g., atomic compare-and-swap operations or locks for mutual exclusion, are ubiquitous, but hard to compose. Futures and async/await [Tasirlar and Sarkar 2011] are abstractions for conveniently handling and composing asynchronous tasks. Actors [Agha 1986] are a message passing-based programming model for loosely-coupled distributed and concurrent event-driven applications. Software Transactional Memory [Shavit and Touitou 1995] schedules arbitrary, composable [Ni et al. 2007] read/write transactions, Reagents [Turon 2012] are abstractions for implementing data structures that can be safely composed into concurrent algorithms. Both use "optimistic locking" which is ill-named since transactions are executed speculatively *without* locking accessed variables, but are only committed if no race conditions occurred until they complete and otherwise aborted. Compared to these solutions, MV-RP provides concurrency management integrated into a fully declarative and dynamically composable programming model.

Scheduling Algorithms from Databases. Most research into scheduling algorithms has been conducted in the scope of database transactions. MV-RP is build upon MVCC and C2PL, which both originated from databases [Bernstein et al. 1986]. We discussed their pure forms extensively throughout the paper, but several variations exist. MVCC can be used in different ways to provide weaker guarantees than serializability, e.g., snapshot isolation [Berenson et al. 1995]. C2PL is one of many locking algorithms, all of which provide strict serializability but with different requirements and properties. In general, 2PL (two-phase locking) states that if locks are acquired in any order but not released during a growing phase, and then released but not acquired during a shrinking phase, strict serializability is achieved. In its rudimentary form, 2PL is prone to deadlocks, so it requires aborts. In Ordered 2PL, locks must be acquired during the growing phase following a fixed global order. This avoids deadlocks and thus provides abort-free execution, but is inapplicable to RP because the dynamic DG does not allow transactions to adhere to a fixed locking order. Tree locking [Lanin and Shasha 1990] uses a similar approach, replacing the two-phase restriction by moving a locked interval along every node on paths in trees as the locking order. DAG locking [Chaudhri and Hadzilacos 1995] generalizes tree locking to directed acyclic graphs, such as the RP DG. Both tree and DAG locking are special cases of domination locking [Golan-Gueta et al. 2011] for yet more general graphs. None of these are applicable to RP either though, because they require "inwards" graph traversals from all predecessors of a node to that node, whereas RP change propagation traverses outwards from each changed node to all successors.

7 CONCLUSION

We propose a language model for thread-safe RP based on abort-free strict serializability to ensure correct behavior under concurrency. We present a corresponding scheduler, integrated into the language runtime, and prove its correctness. The implementation of our approach hides the complexity of concurrency management from developers and is available as an extension of the REScala programming language. Our evaluation shows that applications scale with multi-threading and exhibit acceptable performance overhead.

ACKNOWLEDGEMENTS

This work was supported by the European Research Council (ERC: PACE advanced grant 321217), German Research Foundation (DFG: SA 2918/2-1, and project A2 within CRC 1053 MAKI) and LOEWE initiative in Hessen, Germany (HMWK: NICER).

REFERENCES

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

Roberto M Amadio, Gérard Boudol, Frédéric Boussinot, and Ilaria Castellani. 2006. Reactive concurrent programming revisited. *Electronic Notes in Theoretical Computer Science* 162 (2006), 49–60.

Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4, Article 52 (2013), 34 pages. https://doi.org/10.1145/2501654.2501666

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Björn B. Brandenburg, and Rodrigo Rodrigues. 2015. iThreads: A Threading Library for Parallel Incremental Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 645–659. https://doi.org/10.1145/2694344.2694371

Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for Incremental Computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/2038916.2038923

Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. 2010. CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days*.

Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. 2011. Two for the Price of One: A Model for Parallel and Incremental Computation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 427–444. https://doi.org/10.1145/2048066.2048101

Vinay K. Chaudhri and Vassos Hadzilacos. 1995. Safe Locking Policies for Dynamic Databases. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '95)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/212433.212464

Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-value Language. In *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06)*. Springer-Verlag, Berlin, Heidelberg, 294–308. https://doi.org/10.1007/11693024_20

Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (June 2012), 62 pages. https://doi.org/10.1145/2187671.2187677

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 411–422. https://doi.org/10.1145/2491956.2462161

Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. *Thread-Safe Reactive Programming.* Technical Report.

Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 361–376. https://doi.org/10.1145/2660193.2660240

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. *SIGPLAN Not.* 32, 8 (Aug. 1997), 263–273. https://doi.org/10.1145/258949.258973

Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114–131. https://doi.org/10.1145/857076.857078

Guy Golan-Gueta, Nathan Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. 2011. Automatic Fine-grain Locking Using Shape Properties. *SIGPLAN Not.* 46, 10 (Oct. 2011), 225–242. https://doi.org/10.1145/2076021.2048086

Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. 2007. A Proposal for Parallel Self-adjusting Computation. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP '07)*. ACM, New York, NY, USA, 3–9. https://doi.org/10.1145/1248648.1248651

Li Haoyi. 2016. Scala.Rx Web site. https://github.com/lihaoyi/scala.rx *(Web)*.

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

C. A. R. Hoare. 1985. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Alan Jeffrey. 2013. Functional Reactive Programming with Liveness Guarantees. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.

1145/2500365.2500584

Tetsuo Kamina and Tomoyuki Aotani. 2018. Harmonizing Signals and Events with a Lightweight Extension to Java. *CoRR* abs/1803.10199 (2018). arXiv:1803.10199 http://arxiv.org/abs/1803.10199

Neelakantan R. Krishnaswami. 2013. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 221–232. https://doi.org/10.1145/2500365.2500588

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order Functional Reactive Programming in Bounded Space. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 45–58. https://doi.org/10.1145/2103656.2103665

Vladimir Lanin and Dennis Shasha. 1990. *Tree locking on changing trees.* Technical Report. New York University, Department of Computer Science, Courant Institute of Mathematical Science.

Jesse Liberty and Paul Betts. 2011. *Programming Reactive Extensions and LINQ* (1st ed.). Apress, Berkely, CA, USA.

Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. 2010. Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *Objects, Models, Components, Patterns*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–60.

Ingo Maier and Martin Odersky. 2012. *Deprecating the Observer Pattern with Scala.React.* Technical Report.

Ingo Maier and Martin Odersky. 2013. Higher-Order Reactive Programming with Incremental Lists. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 707–731.

Alessandro Margara and Guido Salvaneschi. 2014. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/2611286.2611290

A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (July 2018), 689–711. https://doi.org/10.1109/TSE.2018.2833109

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:26. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1

Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1229428.1229442

John Peterson, Valery Trifonov, and Andrei Serjantov. 2000. Parallel Functional Reactive Programming. In *Practical Aspects of Declarative Languages*, Enrico Pontelli and Vítor Santos Costa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–31.

José Proença and Carlos Baquero. 2017. Quality-Aware Reactive Programming for the Internet of Things. In *Fundamentals of Software Engineering*, Mehdi Dastani and Marjan Sirjani (Eds.). Springer International Publishing, Cham, 180–195.

Aleksandar Prokopec, Philipp Haller, and Martin Odersky. 2014. Containers and Aggregates, Mutators and Isolates for Reactive Programming. In *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2637647.2637656

Stefan Ramson and Robert Hirschfeld. 2017. Active Expressions: Basic Building Blocks for Reactive Programming. *CoRR* abs/1703.10859 (2017). arXiv:1703.10859 http://arxiv.org/abs/1703.10859

Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 55–68. https://doi.org/10.1145/2661136.2661140

Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014a. An Empirical Study on Program Comprehension with Reactive Programming. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 564–575. https://doi.org/10.1145/2635868.2635895

Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014b. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2577080.2577083

Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: A Functional Reactive Programming Language for Small-scale Embedded Systems. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. ACM, New York, NY, USA, 36–44. https://doi.org/10.1145/2892664.2892670

Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 204–213. https://doi.org/10.1145/224964.224987

Sagnak Tasirlar and Vivek Sarkar. 2011. Data-Driven Tasks and Their Implementation. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP '11)*. IEEE Computer Society, Washington, DC, USA, 652–661. https://doi.org/10.1109/ICPP.2011.87

Aaron Turon. 2012. Reagents: Expressing and Composing Fine-grained Concurrency. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/2254064.2254084

Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA (Nov. 2018). https://doi.org/10.1145/3276499